Scalable Query-based Faceted Search on top of SPARQL Endpoints for Guided and Expressive Semantic Search

Joris Guyonvarch

Scalable Query-based Faceted Search on top of SPARQL Endpoints for Guided and Expressive Semantic Search

Master research Internship report
June 2013

Author:
Joris Guyonvarch

Supervisors:
Sébastien Ferré
Mireille Ducassé

Research Team:
Logical Information Systems (LIS)

Contents

List of Figures

List of Tables

1 Introduction

The internship took place at IRISA[1]. IRISA is a mixed research unit in computer science, signal processing, image processing and robotics. It is composed of 700 people, 35 teams and 7 departments. At IRISA, the internship took place in the Logical Information Systems (LIS) team.

The Web of Documents is composed of structured pages that are not meaningful to machines. Searching in the Web of Documents is generally processed by keywords. However, the Web of Data [12] formalizes data in order to let machines understand and retrieve precise information. Every concept in the Web of Data is identified by a URI[2]. Information is structured as basic sentences called triples. Each triple contains a subject, a predicate and an object. SPARQL [1, 2] is the standard query language for querying this structured information. SPARQL is expressive and its syntax is similar to SQL. However, SPARQL queries are not intended to be written by casual users. A number of works have adopted keyword search for the Web of data [7, 15], but the Web of Data can be explored in a more user-friendly way. On one hand, the query language can be more natural but still formal. A usability study [14] compared several search systems from the less formal to the more formal one. On the other hand, faceted search [21] allows a progressive search of information in which users explore the result set with the application of successive refinements.

Sewelis [9] is a search system that combines those user-friendly concepts. Sewelis uses a formal language that looks natural, and queries are built incrementally. Sewelis is both *safe* and *complete*. Safeness prevents the navigation from running into dead-ends i.e. empty results. Completeness allows the navigation to reach every query that is not a dead-end. However, Sewelis is not yet scalable to every dataset. For example, DBpedia is too large to be searched through Sewelis. DBpedia [4] contains 1.89 billion triples of structured information extracted automatically from Wikipedia and other sources.

We introduce in this report a system inspired from Sewelis that is scalable to DBpedia, Scalewelis. The scalability of a search system on the Semantic Web is an important criterion that have been identified by several search systems [10, 17, 24, 14]. Scalewelis connects to SPARQL endpoints. For example, to access information about DBpedia, Scalewelis can use the DBpedia SPARQL endpoint[3]. Because of the utilization of SPARQL endpoints to access data, Scalewelis is independent from data storage. Moreover, Scalewelis uses up to date data. However, because of data independence, Scalewelis faces limitations in optimizing operations on data.

The report is organized as follows. Section 2 first categorizes existing

[1] Institut de Recherche en Informatique et Systèmes Aléatoires http://www.irisa.fr/
[2] Universal Resource Identifier
[3] http://dbpedia.org/sparql/

search systems for the Semantic Web and discusses their scalability issues. Section 3 introduces a language to search for information. Section 4 describes how queries of this language are created incrementally. Section 5 presents an implementation of a query-based search system based on our language. Section 6 tests the implementation on large datasets. Finally, Section 7 concludes and opens up perspectives.

2 Related Work

The bibliographic study preceding the internship defined a set of important criteria based on previous research [10, 17, 24, 14]. Those criteria are usability, expressivity, scalability, portability, flexibility and the formal degree of a query language. We used those criteria to classify existing search systems into categories. We identified two principal categories of search systems: the ones requiring one query in the overall process and the ones leading progressively to the results with the application of facets. In querying search, we distinguished free input from guided input: free input querying systems are based either on keywords (Swoogle [7]) or on a question (SQUALL [8], Semplore [24], Freya [6] PowerAqua [16], SWIP [20]), whereas guided input querying systems are based either on a question (Ginseng [14]) or on a graph (Semantic Crystal [14]). In faceted search, we distinguished set-based methods (VisiNav [10], Ontogator [17]) from graph-based methods (gFacet [11], FacetMap [22]) and from query-based method (Sewelis [9]). The next two paragraphs explore scalability issues of existing search systems.

Concerning querying search, SQUALL is a controlled natural language that is translated into SPARQL covering most of SPARQL. SQUALL shares the same scalability issues as SPARQL. Semplore allows to search with a formal query containing keywords and facets, it needs an initialization that builds an incremental index. On DBpedia, Semplore takes approximately 30 seconds to build its index and, then, each question is answered in less than 0.5 second. PowerAqua is an ontology-based question answering system. PowerAqua takes 20 seconds on average to compute a query on DBpedia. FREyA is an interactive Natural Language Interface for querying ontologies that asks for clarifications to the user. FREyA initializes itself in 50 hours on DBpedia and takes 36 seconds on average per interaction i.e. question and clarifications.

Concerning faceted search, Ontogator is a set-based faceted search system, it allows transformations on the result set such as filtering or applying intersection and union. Ontogator takes 3.5 seconds on average to answer on a dataset containing 275,707 categories and 2,300,000 items. More expressive than Ontogator, FacetMap is a graph-based faceted search system fully oriented to facet visualization: facets are organized as a tree and each facet contains a set of items. FacetMap can be used with a grid of screen so that

the visualization can scale to bigger datasets. A user-study evaluates the average task time between 250 and 300 seconds. More expressive than the two precedent systems, Sewelis is a faceted search system based on query transformations to lead the search at a syntactic level. However, Sewelis does not scale to datasets above 100,000 triples.

3 Query Language

In Sewelis, the query language is called LISQL[4]. LISQL queries are semi-natural to ease reading for casual users. Moreover, the query is built incrementally: the search begins with an initial query that asks for every class and every property. Then, Sewelis provides successive valid query elements to complete the query and their application on the query guides the user to explore data without dead-end. In Scalewelis, we introduce LISQL2 that is a new version of LISQL.

LISQL2 takes inspiration from SQUALL [8]. SQUALL is a controlled natural language that covers most of SPARQL but it provides no guidance. LISQL2 is a sub-language of SQUALL for which the incremental construction is defined. The incremental construction is reviewed in Section 4. LISQL2 is thus a simplified natural language, it is semi-natural in the sense of natural but simplified and, second, it is built incrementally so that users do not have to write queries but only to select query elements. Compared to LISQL, there are two main improvements. First, LISQL2 is more readable because of a more natural grammar. The LISQL2 grammar separates any sentence into two different levels, a verb phrase and a noun phrase, instead of one level. Second, the definition of a variable is separated from its use.

Scalewelis uses an external SPARQL endpoint to query data with SPARQL queries, unlike Sewelis which uses an internal triple store and query engine. In order to access the SPARQL endpoint, we define a SPARQL semantics for LISQL2.

3.1 Concrete Syntax

The concrete syntax of LISQL2 follows fifteen production rules (Figure 1). Those rules describe the production of each nonterminal symbol into a list of terminal and nonterminal symbols. Nonterminal symbols are typed in italic whereas terminal ones are typed in bold. Some of the nonterminal symbols are underlined: that means that they can get the focus (Section 4.1). *(c1)* produces a sentence with both a subject and a verb phrase. *(c2)* produces a non specified subject whereas *(c3)* produces an URI subject. *(c4)* produces a noun affectation, *(c5)* produces a transitive verb applied to a noun phrase, *(c6)* produces a conjunction of verb phrases, *(c7)* produces a disjunction of

[4]LIS Query Language

verb phrases and *(c8)* produces the negation of a verb phrase. *(c9)* produces a non specified noun whereas *(c10)* produces a class noun. *(c11)* produces a property whereas *(c12)* produces the inverse of a property. *(c13)* produces a node, *(c14)* produces a noun with an optional variable and an optional verb phrase, *(c15)* produces a variable.

Nonterminal symbols represent sentences, subjects, verb phrases, nouns, transitive verbs, noun phrases, URIs, Classes, Properties, Nodes and Variables. Some of the nonterminal symbols are not described by any production rule: *URI*, *Class*, *Property*, *Node* and *Variable*. A *URI* is a universal resource identifier, a *Class* is a class *URI*, a *Property* is a property *URI*, a *Node* is either a literal or a *URI*, and finally a *Variable* allows the language to build graph patterns instead of simple tree patterns. Graph patterns correspond to the use of references. For example, graph patterns allows to answer to the following question: *Give me every films in which* ***its*** *director is starring.* The reference is in bold in the question.

In order to illustrate the concrete syntax, we use a simple example covering the most important constructions of LISQL2: *What is a book and has author a writer that has nationality Russian?*. This query is derived from the *Sentence* axiom using a succession of production rules (Figure 2).

3.2 Abstract Syntax

The abstract syntax of LISQL2 represents its internal representation (Figure 3), where each rule noted (c_i) in the concrete syntax is related to the rule noted (a_i) in the abstract syntax. Consequently, the internal representation of sentences, subjects, verb phrases, nouns, transitive verbs and noun phrases are presented. Optional elements are the same in the abstract syntax as in the concrete syntax, as well as the underlined nonterminal symbols.

A sentence *(a1)* is composed of a subject and a verb phrase. A subject is either non specified *(a2)* or a URI *(a3)*. A verb phrase is either a type *(a4)*, a restriction *(a5)*, a conjunction *(a6)*, a disjunction *(a7)* or a negation *(a8)*. A noun is either a thing *(a9)* or a class *(a10)*. A transitive verb is either a property *(a11)* or an inverse property *(a12)*. Finally, a noun phrase is either a node *(a13)*, an extended type *(a14)* or a variable *(a15)*.

The abstract syntax tree of the previous example, *What is a book and has author a writer that has nationality Russian?*, is build following the abstract syntax of LISQL2 (Figure 4). Operators are represented in boxes whereas operands are represented in double rounded boxes. Types of operators are indicated on edges.

3.3 Semantics

The Semantics of LISQL2 is obtained by translating the abstract syntax of LISQL2 into SPARQL. Each production rule of the concrete and abstract

$$
\begin{aligned}
\mathit{Sentence} &\rightarrow \underline{\mathit{Subject}}\ \underline{\mathit{VerbPhrase}}\ \mathbf{?} && (c1)\\
\mathit{Subject} &\rightarrow \mathbf{What} && (c2)\\
&\mid \mathit{URI} && (c3)\\
\mathit{VerbPhrase} &\rightarrow \mathbf{is\ a}\ \mathit{Noun} && (c4)\\
&\mid \mathit{TransitiveVerb}\ \underline{\mathit{NounPhrase}} && (c5)\\
&\mid \underline{\mathit{VerbPhrase}}\ \mathbf{and}\ \underline{\mathit{VerbPhrase}} && (c6)\\
&\mid \underline{\mathit{VerbPhrase}}\ \mathbf{or}\ \underline{\mathit{VerbPhrase}} && (c7)\\
&\mid \mathbf{not}\ \underline{\mathit{VerbPhrase}} && (c8)\\
\mathit{Noun} &\rightarrow \mathbf{thing} && (c9)\\
&\mid \mathit{Class} && (c10)\\
\mathit{TransitiveVerb} &\rightarrow \mathbf{has}\ \mathit{Property} && (c11)\\
&\mid \mathbf{is}\ \mathit{Property}\ \mathbf{of} && (c12)\\
\mathit{NounPhrase} &\rightarrow \mathit{Node} && (c13)\\
&\mid \mathbf{a}\ \mathit{Noun}\ (\mathbf{[}\mathit{Variable}\mathbf{]})?\ (\mathbf{that}\ \underline{\mathit{VerbPhrase}})? && (c14)\\
&\mid \mathit{Variable} && (c15)
\end{aligned}
$$

Figure 1: LISQL concrete syntax

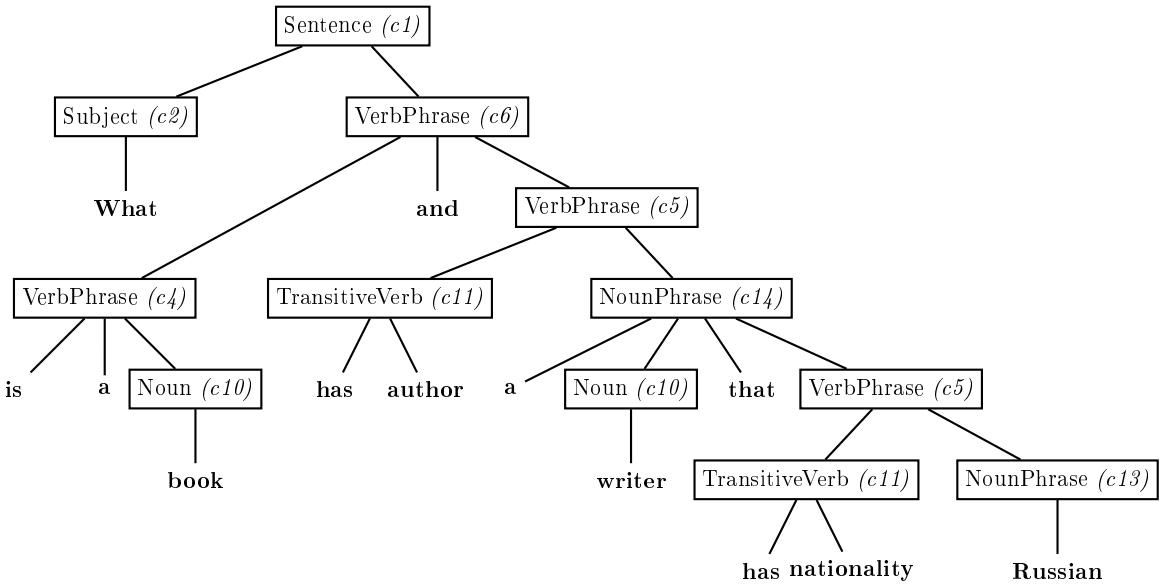

Figure 2: Concrete syntax tree of the query *What is a book and has author a writer that has nationality Russian?*

$$
\begin{aligned}
\mathit{Sentence} &\rightarrow \textbf{Sentence}(\underline{\mathit{Subject}},\ \underline{\mathit{VerbPhrase}}) && (a1)\\
\mathit{Subject} &\rightarrow \textbf{What}() && (a2)\\
&\mid \textbf{URI}(\mathit{URI}) && (a3)\\
\mathit{VerbPhrase} &\rightarrow \textbf{Type}(\mathit{Noun}) && (a4)\\
&\mid \textbf{Restriction}(\mathit{TransitiveVerb},\ \underline{\mathit{NounPhrase}}) && (a5)\\
&\mid \textbf{Conjunction}(\underline{\mathit{VerbPhrase}},\ \underline{\mathit{VerbPhrase}}) && (a6)\\
&\mid \textbf{Disjunction}(\underline{\mathit{VerbPhrase}},\ \underline{\mathit{VerbPhrase}}) && (a7)\\
&\mid \textbf{Negation}(\underline{\mathit{VerbPhrase}}) && (a8)\\
\mathit{Noun} &\rightarrow \textbf{Thing}() && (a9)\\
&\mid \textbf{Class}(\mathit{Class}) && (a10)\\
\mathit{TransitiveVerb} &\rightarrow \textbf{Property}(\mathit{Property}) && (a11)\\
&\mid \textbf{InvProperty}(\mathit{Property}) && (a12)\\
\mathit{NounPhrase} &\rightarrow \textbf{Node}(\mathit{Node}) && (a13)\\
&\mid \textbf{ExtendedType}(\mathit{Noun},\ \mathit{Variable}?,\ \underline{\mathit{VerbPhrase}}?) && (a14)\\
&\mid \textbf{Variable}(\mathit{Variable}) && (a15)
\end{aligned}
$$

Figure 3: LISQL abstract syntax

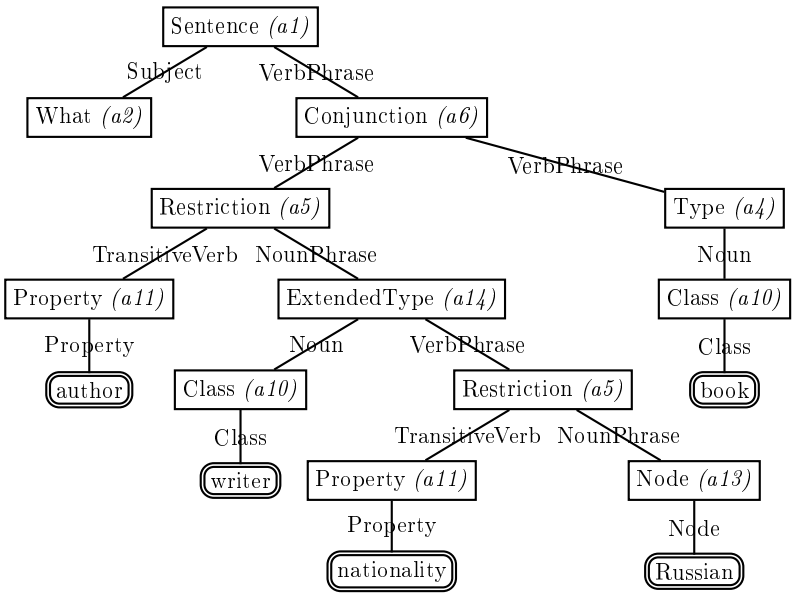

Figure 4: Abstract syntax tree of the query *What is a book and has author a writer that has nationality Russian?* corresponding to the concrete syntax of Figure 2

($s1$) **SELECT DISTINCT** v *Vars(VerbPhrase)* **WHERE {** [*VerbPhrase*] [*Subject*]**}**

for $v = \begin{cases} \textbf{?What} & \text{if } [Subject] = \textbf{?What}, \\ \epsilon & \text{otherwise.} \end{cases}$

($s2$) **?What**

($s3$) *URI*

($s4$) $\lambda x.([Noun]\, x)$

($s5$) $\lambda x.([NounPhrase]\, \lambda y.([Transitive\, Verb]\, x\, y))$

($s6$) $\lambda x.(([VerbPhrase_1]\, x)\, ([VerbPhrase_2]\, x))$

($s7$) $\lambda x.(\textbf{\{}[VerbPhrase_1]\, x\textbf{\}}\ \textbf{UNION}\ \textbf{\{}[VerbPhrase_2]\, x\textbf{\}})$

($s8$) $\lambda x.(\textbf{FILTER NOT EXISTS \{}([VerbPhrase]\, x)\textbf{\}})$

($s9$) $\lambda x.(\epsilon)$

($s10$) $\lambda x.(x\ \textbf{a}\ Class\ \textbf{.})$

($s11$) $\lambda x.\lambda y.(x\ Property\ y\ \textbf{.})$

($s12$) $\lambda x.\lambda y.(y\ Property\ x\ \textbf{.})$

($s13$) $\lambda f.(f\ Node)$

($s14$) $\lambda f.((fv)\ ([Noun]\, v)\, ([VerbPhrase]\, v)?)$

for $v = \begin{cases} Var & \text{if given,} \\ \text{a fresh variable} & \text{otherwise.} \end{cases}$

($s15$) $\lambda f.(f\ Var)$

Figure 5: LISQL2 semantics

syntax of LISQL2 has an equivalent translation rule to SPARQL noted *(s_i)*. Figure 5 shows those rules inspired from SQUALL [8]. Translation rules use the lambda calculus formalism [3]: every variable is bound and substitutions are applied on the call of nonterminal symbols between brackets.

A sentence *(a1)* is translated to the rule *(s1)*. There are two replacements in *(s1)*. First, v is replaced to the variable *?What* if the subject is non specified, to nothing otherwise. Second, *Vars(VerbPhrase)* is replaced to the list of LISQL2 variables that belong to the verb phrase. The keyword *SELECT DISTINCT* in *(s1)* means that results are not duplicated. A non specified subject *(a2)* is translated to *?What (s2)*, whereas a URI subject *(a3)* is translated to this URI *(s3)*.

Verb phrases translation rules bind a variable corresponding to the subject of the verb phrase. A type verb phrase *(a4)* is translated to the rule *(s4)* that applies the subject to the noun from the rule *(a4)*. A restriction verb phrase *(a5)* is translated to the rule *(s5)* that applies to the noun phrase from the rule *(a5)* the application of both the subject and a free binding

to the transitive verb from the rule *(a5)*. A conjunction *(a6)* is translated to the rule *(s6)* that applies the subject to both the verb phrases from the rule *(a6)*. A disjunction *(a7)* is translated to the rule *(s7)* that applies the subject to both the verb phrases from the rule *(a7)* and inserts a SPARQL union between them. A negation *(a8)* is translated to the rule *(s8)* that applies the subject to the verb phrase from the rule *(a8)* and filters the solutions discarding values bound to this verb phrase. It does not generate any SPARQL binding but only filters solutions. For example, the query *What not is a writer?* does not generate any binding.

Noun translation rules bind a variable corresponding to the subject on which the subject is applied. A thing *(a9)* is translated to nothing *(s9)*. A class *(a10)* is translated to the rule *(s10)* that restricts the subject to be an instance of the class. The ending dot at the end of the rule *(s10)* separates SPARQL sentences. A property *(a11)* is translated to the rule *(s11)* that applies a SPARQL property from x to y. An inverse property *(a12)* is translated to the rule *(s12)* that applies a SPARQL property from y to x. x is the variable associated to the domain subject whereas y is the variable associated to the range subject.

Noun phrase translation rules bind a function that is applied on the noun. A node *(a13)* is translated to the rule *(s13)* that applies the function to the node from the rule *(a13)*. An extended type *(a14)* is translated to the rule *(s14)*. In the rule *(s14)*, v is replaced to the variable of the extended type if given, to a fresh variable otherwise. In the rule *(s14)*, the function is applied to v, v is applied to the noun from the rule *(a14)* and to the verb phrase from the rule *(a14)* if given. A variable *(a15)* is translated to the rule *(s15)* that applies the function to the variable from the rule *(a15)*.

Returning to our example *What is a book and has author a writer that has nationality Russian?*, we can generate the SPARQL semantics (Figure 6) using translation rules (Figure 5). At intermediate step, only the sentence *(s1)*, its subject *(s2)* and its verb phrase *(s6)* are translated.

4 Incremental Construction

LISQL2 queries can be built incrementally, we thereby formalize the position in a LISQL2 query from where query elements are added and that is called the focus. We propose several methods to compute query elements that can be inserted into a query. Those methods are then tested in Section 6.

We call increments every query element that can be inserted into a query. Our approach consists in giving users an initial LISQL2 query and then ask to complete this query with proposed increments. Given a query, there is a finite number of increments that can be used to complete this query. Increments belong to LISQL2 and are classes, properties, inverse properties, variables and nodes. Increments can be seen as special facets whose purpose

Intermediate step

```
SELECT DISTINCT ?What WHERE {
   [is a book] ?What
   [has author a writer that nationality Russian] ?What }
```

Final step

```
SELECT DISTINCT ?What WHERE {
   ?What a book .
   ?What author ?v1 .
   ?v1 a writer .
   ?v1 nationality Russian }
```

Figure 6: Semantics of the query *What is a book and has author a writer that has nationality Russian?* corresponding to the abstract syntax of Figure 4

is to complete the query. In order to make the query grow, we define a concept that gives access to every interesting position in a query: the focus.

4.1 Focus

The focus is a point in the query from where increments can be added. As the query can grow from different places, the focus has to be movable. A naive solution to define the focus is to interpret it as a path in the query, that path would give the desired place in the query and the focus would move by changing that path. But if we choose this solution, we have to ensure that the path is always correct in the query: the path and the query are two strongly linked concepts working independently. A better solution to define the focus is to combine both the query and the path into a unique concept: this is possible by formalizing the context of the focus.

The abstract syntax of the focus (Figure 7) follows the same conventions as grammars presented above in this paper. These translation rules complete the 15 production rules previously presented. The focus is formally defined so that it allows inductive definitions. We define the focus as a pair of two elements: a LISQL2 sub-query and the context of this sub-query. The context contains the whole information about the sub-query so that the sub-query and its context are sufficient to build back the global LISQL2 query. We get in addition a position in the global query from where the query can grow.

The first three production rules from *(a16)* to *(a18)* present the three different forms of the focus. Those rules have the same structure: the first term in italic is the sub-query and the second one is the context of that sub-query. From those rules we see that the focus is only applied on either

$$
\begin{aligned}
\mathit{Focus} &\rightarrow \textbf{VerbPhraseFocus}(\mathit{VerbPhrase}, \mathit{VerbPhraseCtxt}) && (a16)\\
&\mid \textbf{NounPhraseFocus}(\mathit{NounPhrase}, \mathit{NounPhraseCtxt}) && (a17)\\
&\mid \textbf{SubjectFocus}(\mathit{Subject}, \mathit{SubjectCtxt}) && (a18)\\
\mathit{VerbPhraseCtxt} &\rightarrow \textbf{SentenceCtxt}(\mathit{Subject}) && (a19)\\
&\mid \textbf{LeftConjunctionCtxt}(\mathit{VerbPhrase}, \mathit{VerbPhraseCtxt}) && (a20)\\
&\mid \textbf{RightConjunctionCtxt}(\mathit{VerbPhrase}, \mathit{VerbPhraseCtxt}) && (a21)\\
&\mid \textbf{LeftDisjunctionCtxt}(\mathit{VerbPhrase}, \mathit{VerbPhraseCtxt}) && (a22)\\
&\mid \textbf{RightDisjunctionCtxt}(\mathit{VerbPhrase}, \mathit{VerbPhraseCtxt}) && (a23)\\
&\mid \textbf{NegationCtxt}(\mathit{VerbPhraseCtxt}) && (a24)\\
&\mid \textbf{ExtendedTypeCtxt}(\mathit{Noun}, \mathit{Var?}, \mathit{NounPhraseContext}) && (a25)\\
\mathit{NounPhraseCtxt} &\rightarrow \textbf{RestrictionCtxt}(\mathit{Transitive\,Verb}, \mathit{VerbPhraseCtxt}) && (a26)\\
\mathit{SubjectCtxt} &\rightarrow \textbf{SentenceCtxt}(\mathit{VerbPhrase}) && (a27)
\end{aligned}
$$

Figure 7: Focus and context abstract syntax

a verb phrase, a noun phrase or a URI. That is consistent with the concrete and abstract grammars of Figure 1 and 3 where underlined sub-queries are the ones that can be on focus.

A context is either a verb phrase context (from *(a19)* to *(a25)*), a noun phrase context *(a26)* or a subject context *(a27)*. The purpose of the context is to build back the global query with the sub-query. In either the concrete or the abstract syntax of LISQL2 (Figures 1, 3); there are as many different elements possibly on focus as there are different contexts. Indeed, any sub-query that can be on focus needs a context to keep trace of the global query. For example, a conjunction verb phrase *(a6)* can have either the left verb phrase or the right verb phrase on focus. Consequently, if the focus is on the left verb phrase, the focus contains two elements: the previous left verb phrase and a verb phrase context, that is a right conjunction context *(a21)* and contains the right verb phrase and the verb phrase context of the conjunction and so on. In fact, every focus and context production rule is directly deduced from the rules in the abstract syntax of LISQL2 and the sub-queries that can be on focus.

From the previous query example, *What is a book and has author* ***a writer that has nationality Russian****?*, we arbitrarily set up the focus on the Russian author (Figure 8). The abstract syntax tree of this focus follows the abstract syntax of both LISQL2 and the focus (Figures 3, 7) as it is indicated with rule numbers. The two parts of the focus are identifiable in the tree: the noun phrase (*a writer that nationality Russian*) grows downward and the noun phrase context (*What is a book and author ... ?*) grows upward.

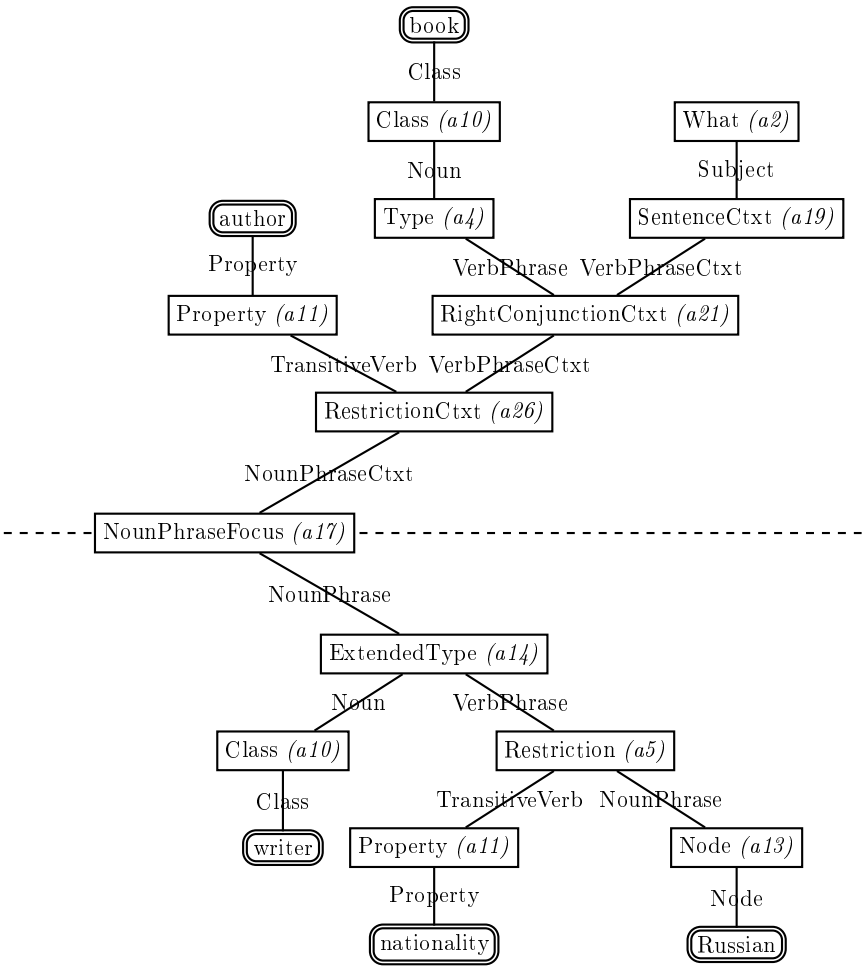

Figure 8: Abstract syntax tree of the focus *What is a book and has author* ***a writer that has nationality Russian****?*

($s16$) [*VerbPhraseContext*] *VerbPhrase*

($s17$) [*NounPhraseContext*] *NounPhrase*

($s18$) [*SubjectContext*] *URI*

($s19$) $\lambda vp.$(**Sentence**(*Subject*, *vp*))

($s20$) $\lambda vp.$([*VerbPhraseCtxt*] **Conjunction**(*VerbPhrase*, *vp*))

($s21$) $\lambda vp.$([*VerbPhraseCtxt*] **Conjunction**(*vp*, *VerbPhrase*))

($s22$) $\lambda vp.$([*VerbPhraseCtxt*] *vp*)

($s23$) $\lambda vp.$([*VerbPhraseCtxt*] *vp*)

($s24$) $\lambda vp.$([*VerbPhraseCtxt*] *vp*)

($s25$) $\lambda vp.$([*NounPhraseCtxt*] **ExtendedType**(*Noun*, *Var?*, *vp*))

($s26$) $\lambda np.$([*VerbPhraseCtxt*] **Restriction**(*TransitiveVerb*, *np*))

($s27$) $\lambda s.$(**Sentence**(s, *VerbPhrase*))

Figure 9: Focus and context translation

In order to get answers from a focus, we first translate the focus into a LISQL2 query (Figure 9). In this figure, we use the same conventions as with the precedent semantics. The first three rules, from *(s16)* to *(s18)*, apply the sub-query to the context. Verb phrase context rules, from *(s19)* to *(s25)*, build from a verb phrase: a sentence in *(s19)*, a conjunction in *(s20)* and *(s21)*. From *(s22)* to *(s24)*, disjunctions and negations are forgotten and nothing more than verb phrases in argument are produced. In *(s25)*, an extended type is produced from the verb phrase in argument. The noun phrase context rule *(s26)* build from a noun phrase a restriction. Finally, the subject context rule *(s27)* build from a subject a sentence. Once we have translated the focus into a LISQL query, we translate the LISQL query into SPARQL (Figure 5).

As in Sewelis [9], disjunctions and negations containing the focus are deleted for the translation of the focus into a LISQL2 query. If a disjunction contains the focus, this disjunction is replaced by its verb phrase leading to the focus. If a negation contains the focus, this negation is replaced by its verb phrase. For example, the focus *What is a man and not **is a scientist?*** is translated into the query *What is a man and is a scientist?*. However, disjunctions and negations contained by the focus are still present LISQL2 query translated from the focus. For example, the focus *What **is a man and not is a scientist?*** is translated into the query *What is a man and not is a scientist?*.

We saw above that the focus is translatable into a LISQL2 query. More-

over, the translation of a focus into a LISQL2 query also promotes an element that defines the actual level of the query on focus. That promotion is not presented in the Figure. A query level is defined by the variable used as subject in the query. A new subject variable is introduced each time a restriction is crossed. Consequently, each restriction leads to a new level in the query.

If the sub-query on focus is at the root level of the query, then there is two possibilities. First, a URI is specified for the sentence *(a1)* so that this URI is the promoted element. Second, no URI is specified and this is the variable *?What* that is promoted. Else, if the focus is on a noun phrase which is a node, so, the element promoted is that node. Else, the variable that is on the first extended type from the sub-query to the root of the global query is promoted. If there is no defined variable in the extended type, then a new variable, *?This*, is introduced and is promoted. The next section describe how to compute increments from this promoted element on focus.

4.2 Increment Computation

Sewelis computes increments directly from data in order to be *safe* and *complete*. However, computation of increments from data takes time proportionally to the size of datasets. Consequently, Sewelis gets timeouts for the computation of increments on a dataset such as DBpedia.

We explore two methods to compute increments from data: the first one uses every results on focus to compute increments whereas the second one uses only a subset of results on focus to compute increments. Computation of increments is based on results on focus because, as with faceted search, the purpose is to restrict the result set in order to navigate. The result set is restricted in Scalewelis at a syntactic level.

In the following, only increments related to classes, properties and inverse properties are considered. We do not explore in detail variable increments because they are used only by specific queries. Moreover, node increments correspond to results on focus. Since there are too many results on focus, we do not detail node increments.

Increments can be translated to verb phrases in order to be inserted into the query with various operations. If the focus is on a verb phrase, a conjunction *(a6)* is inserted with the previous verb phrase on focus as first operand and with the increment as second operand. If the focus is on an extended type *(a14)*, a conjunction is inserted with the previous verb phrase as first operand and the increment as second operand. However, disjunctions and negations are added to the query at a syntactic level, there are not related to increments.

Method 1: *all the results on focus* Increments are computable from all the results on focus by querying the SPARQL endpoint: the purpose is

to collect every possible increment with a finite number of SPARQL queries. However, this method is not an exact one because the retrieval of increments depends on the endpoint. It is up to it to give a complete answer or not. For example, The SPARQL endpoint of DBpedia[5] does not provide more than 50.000 results for a single query and does not answer to questions it judges too complex, in order to save bandwidth so that the endpoint is more accessible.

Class, property and inverse property increments are computed from results on focus in two steps. First, the focus is translated into a LISQL2 query with an element on focus. Second, the LISQL2 query and the element on focus are translated together to SPARQL in order get both classes, properties and inverse properties. In the following examples, $<Query>$ is the SPARQL sub-query inside the *WHERE* brackets translated from the LISQL2 query and $<This>$ is the element on focus, for example a variable or a URI.

We define three methods to compute increments from all the results on focus (Figure 10). The first method *(C0)* compute increments in a single query. The computation of this query on a SPARQL endpoint results on a list of all correct bindings for the increments in the query. If there are n_1 distinct classes, n_2 distinct properties and n_3 distinct inverse properties, then there are $n_1 * n_2 * n_3$ distinct correct bindings for the query.

In order to escape this combinatorial problem, one query can be used to compute each increment. The second method *(C1)* computes increments in several queries. The first query computes class increments, the second query computes property increments and the third query computes inverse property increments.

Finally, the third method *(C2)* computes increments in a single query. This method escape the combinatorial problem of the method *(C0)* thanks to *BIND* and *UNION* operations in SPARQL. In the SPARQL query, *?incr* is the increment and *?type* is the type of this increment. The type is either class, property or inverse property.

However, the computation of increments from all the results with any of the previous variants does not scale to large amounts of data, as shown by the experiments in Section 6. That is why we introduce a computation of increments with a subset of results on focus.

Method 2: *a subset of results on focus* Computation of increments from a subset of results on focus (Figure 11) takes place in two steps. First, the subset of valid results is collected with the SPARQL query *(V)*. The accuracy limits the number of results to be collected. For example, with an accuracy of 100, the SPARQL endpoint limits the number of returned answers to 100. Second, these collected values are injected to SPARQL queries *(P)* thanks to the SPARQL construct *VALUES* in order to compute

[5] DBpedia endpoint http://dbpedia.org/sparql

(C0) Increment computation with a single query

```
SELECT DISTINCT ?class ?prop ?invProp WHERE {
   <Query>
   <This> a ?class .
   <This> ?prop [] .
   [] ?prop <This> }
```

(C1) Increment computation with several queries

```
SELECT DISTINCT ?class WHERE {
   <Query>
   <This> a ?class }
SELECT DISTINCT ?prop WHERE {
   <Query>
   <This> ?prop [] }
SELECT DISTINCT ?invProp WHERE {
   <Query>
   [] ?invProp <This> }
```

(C2) Increment computation with a single optimized query

```
SELECT DISTINCT ?type ?incr WHERE {
   <Query> .
   { <This> a ?incr BIND("class" AS ?type) }
   UNION {
     <This> ?incr [] BIND("prop" AS ?type) }
   UNION {
     [] ?incr <This> BIND("invProp" AS ?type) } }
```

Figure 10: Increment computation

(V) Values computation

```
SELECT DISTINCT ?This WHERE {
   <Query> } LIMIT accuracy
```

(P) Increment computation from values

```
SELECT DISTINCT ?class WHERE {
   VALUES (?Value) {res1 ... resN}
   ?Value a ?class }
SELECT DISTINCT ?prop WHERE {
   VALUES (?Value) {res1 ... resN}
   ?Value prop [] }
SELECT DISTINCT ?invProp WHERE {
   VALUES (?Value) {res1 ... resN}
   [] invProp ?Value }
```

Figure 11: Partial increment computation

```
SELECT DISTINCT ?class WHERE {
   ?c a [] }
SELECT DISTINCT ?property WHERE {
   [] ?property [] }
```

Figure 12: Increment computation at initialization step

increments from those values.

Note that a not limited accuracy in this method would simulate the behavior of the precedent method because SPARQL variable *?Value* would be bound to all the results. The method using a subset of results on focus to compute increments has better chances to scale to large datasets, as our experiments show in Section 6.

4.3 Navigation Scenario

The focus example, *What is a book and has author* ***a writer that has nationality Russian****?*, is reached by a navigation scenario with successive applications of increments (Figure 13). At least six steps are required to access the desired focus. At initialization step, increments are computed with two queries (Figure 12). The first query computes all the classes whereas the second one computes all and both the properties and inverse properties:

From the initial step, the selection of the *book* class leads to step two, *What* ***is a book****?*, which corresponds to a verb phrase focus *(a12)* with a class

1. *What **is a thing**?*
2. *What **is a book**?*
3. *What is a book and has author **something**?*
4. *What is a book and has **author a writer**?*
5. *What is a book and has author a writer that has nationality **something**?*
6. *What is a book and has author **a writer that has nationality Russian**?*

Figure 13: Navigation scenario from the initial step to: *What is a book and has author **a writer that has nationality Russian**?*

verb phrase *(a2)* and a sentence context *(a15)*. Then, the selection of the *author* property leads to step three, *What is a book and has author **something**?*. Furthermore, the *book* class and the *author* property are grouped into a conjunction and ***something*** is a class noun phrase *(a10)* without any indication about class, variable or verb phrase. From this step to step 6, *What is a book and has author **a writer that has nationality Russian**?*, there are three selections. First, the *writer* class is selected, second, the *nationality* property is selected and, third, the *Russian* URI is selected in the list of results. Moreover, after each of those selections, the focus is moved but this is not detailed in the scenario.

5 Implementation

Scalewelis includes three main modules: the user interface, the LIS engine and the SPARQL endpoint (Figure 14). The user interface and the LIS engine interact with each other and the LIS engine and the SPARQL endpoint interact with each other. Scalewelis is available on a web page[6].

User Interface The user interface (Figure 15) is a layer on top of the LIS engine that orders operations to the LIS engine and displays from the LIS engine information about navigation i.e. query, results and increments. The focus presented in the screenshot is the same as previous examples of this paper except that URIs belong to a domain: *dbo* is used for *http://dbpedia.org/ontology/* and *dbr* is used for *http://dbpedia.org/resource/*. Results are presented in a table with two columns with each line corresponding to an entire result:

[6] http://lisfs2008.irisa.fr/scalewelis/

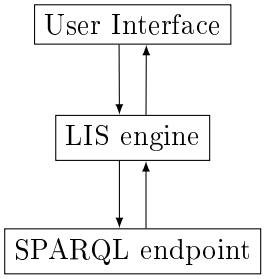

Figure 14: Scalewelis architecture

the first column corresponds to the root variable of the query i.e. books, whereas the second column corresponds to the focus variable of the query i.e. writers. There are 36 results and only the 25 first are presented in this page. Increments are presented in a list. We filtered the list of increments to *dbo* increments. There are 4 class increments, 26 property increments and 12 inverse property increments. If we had not filtered the list of increments, the only visible increments would have been YAGO ones. YAGO [13] provides around 9 million categories corresponding each to classes and properties mixed together like *RussianScienceFictionWriters* or *SaintPetersburgPolytechnicalUniversityAlumni*.

In the user interface, anyone can delete part of the query with *Home* and *Delete*, go back or further with *Undo* and *Redo*, add variables to the query with *Name* and *Unname*, move the focus with *left*, *up*, *right* and *down*, filter increments and results and select one of them to insert in the query.

We built the user interface as a Web Application because those applications do not require any installation from the end-user point of view. The user interface has been written in Java with the Google Web Toolkit (GWT) library: GWT allows the creation of Rich Internet Applications (RIA) from the JAVA language. The code of the user interface contains 1,300 lines of JAVA commentaries excluded.

LIS engine The LIS engine is the core of Scalewelis, it links the user interface to the SPARQL endpoint. The LIS engine handles the focus, the translation from a focus to a LISQL2 query and its focus element, the translation from a LISQL2 query to SPARQL, the computation of valid increments at the focus, the interrogation of SPARQL engines with Jena[7].

The LIS engine is implemented in Scala [19] that merges functional and object oriented programming. The LIS engine is implemented without using accidental nor essential states [18], following Scala recommendations. Moreover, the LIS engine is directly called from the user interface because Scala code is completely compatible with Java code. The code of the LIS engine

[7] http://jena.apache.org/

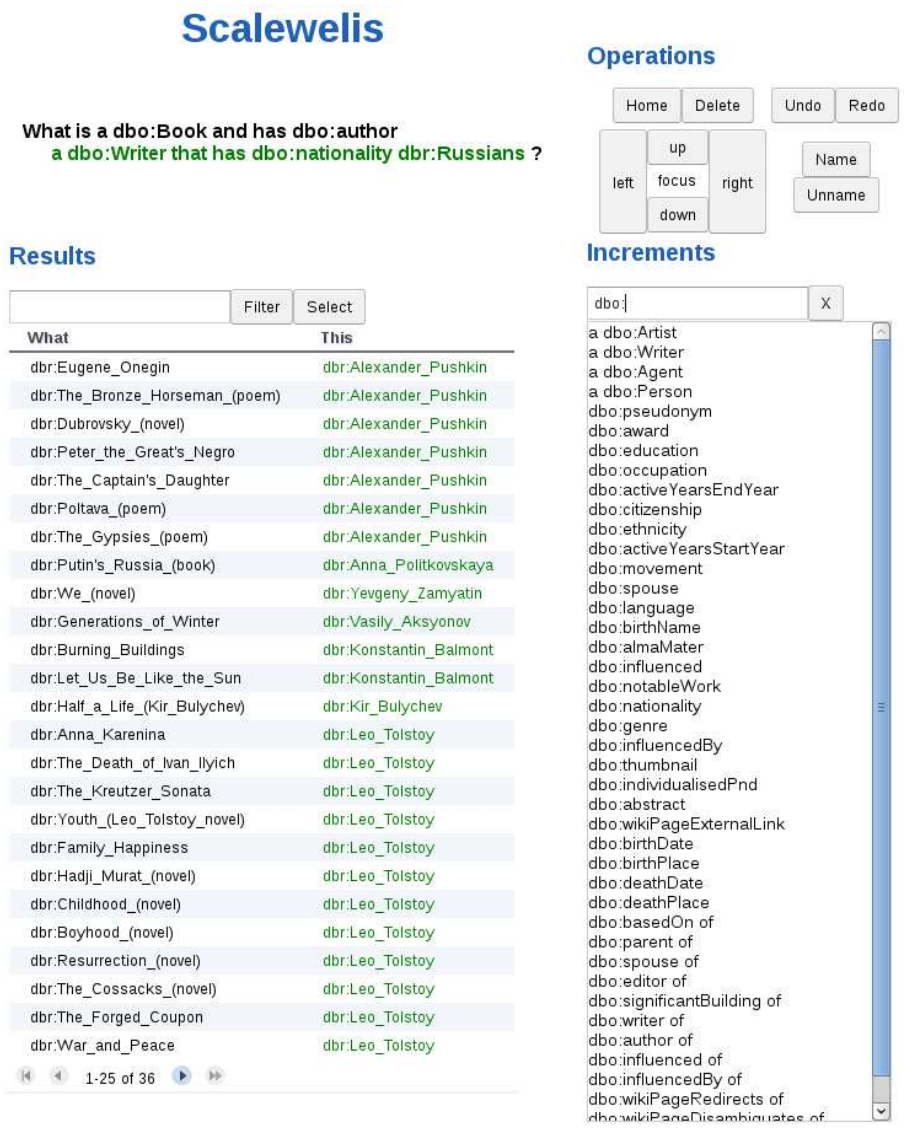

Figure 15: User interface on the DBpedia SPARQL endpoint with focus: *What is a book and has author* ***a writer that has nationality Russian****?*

	Triples	Classes	Properties
DS1	1,200,000	117,012	40
DS2	5,600,000	515,497	40
DS3	15,800,000	1,443,955	40

Table 1: Size of datasets generated from Berlin SPARQL Benchmark [5] to test the navigation scenario

contains 2,500 lines of Scala commentaries excluded.

SPARQL endpoint A SPARQL endpoint is an entry point to a dataset with SPARQL queries. The LIS engine does not maintain datasets but query them in order to get information. The separation between the LIS engine and the data handler allows to connect the LIS engine to any SPARQL endpoints, although the SPARQL endpoint has to implement functions required by the LIS engine. Moreover, the LIS engine uses only one SPARQL engine at a time since querying multiple SPARQL engines is another problem.

6 Experiments

We made preliminary experiments on DBpedia, but we did not conduct our experiments on it for three main reasons. First, querying the SPARQL endpoint on DBpedia involves unstable network time, sometimes the SPARQL endpoint was very slow. Second, the SPARQL endpoint of DBpedia shuts the query down if the computation of results takes too much time or if the SPARQL endpoint judges the query too complex. Third, we could not test the impact of different sizes of the dataset on DBpedia.

We evaluated Scalewelis by measuring the number of increments and the necessary time to compute them on each step of a navigation scenario through six different methods applied on three growing datasets.

In order to experiment on datasets of the same structure but with growing sizes, we used the Berlin SPARQL benchmark [5] to generate triples about vendors offering products and consumers posting reviews about products. As shown in Table 1, this generator allows to specify the number of products in the generated dataset: 3,500 products generate the dataset of 1 million triples, 16,000 products generate the dataset of 5 million triples and 45,000 products generate 15 million triples.

At initialization step, *What **is a thing**?*, increments are computed with two special queries (Figure 12). In fact, in our implementation, we compute only class increments at initialization step because they are sufficient to begin the search and, as we will see in section 6.2.1, property and inverse property increment computation takes far more time than class increment computation and become unacceptable in complete methods. Moreover, there is far

Prefix	Signification
foaf	http://xmlns.com/foaf/0.1/
voc	http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/
countries	http://downlode.org/rdf/iso-3166/countries#
rev	http://purl.org/stuff/rev#
instance	http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/
purldc	http://purl.org/dc/elements/1.1/
instanceSite1	http://www4.wiwiss.fu-berlin.de/bizer/ bsbm/v01/instances/dataFromRatingSite1/

Table 2: Signification of the prefixes used in the navigation scenario

	Navigation	Particularity
P1	partial	10 values
P2	partial	100 values
P3	partial	1,000 values
P4	partial	10,000 values
C1	complete	several queries
C2	complete	one query

Table 3: Methods used to test the navigation scenario

more property increments than class increments and on large datasets the number of properties is so important that it would get the user lost.

The navigation scenario (Figure 16) searches for the name of an American reviewer of a review for a product of type 81, those reviewers having RatingSite1 as publisher. We chose this navigation scenario for two main reasons. First, because the computation time of increments at each step required a time neither too brief nor too long for a human. Second, because this navigation scenario test the most important construction of LISQL2. Table 2 gives the signification of the prefixes used in the navigation scenario.

As shown in Table 3, the navigation scenario is tested on six different methods: four partial methods and two complete methods (figure 3). Partial methods compute increments with growing accuracy with three queries (Figure 11). The first partial method *(P1)* computes increments with 10 values. The second one *(P2)* computes increments with 100 values. The third one *(P3)* computes increments with 1,000 values. The last one *(P4)* computes increments with 10,000 values. The first complete method *(C1)* computes increments with one query for each type of increment i.e. class, property and inverse property increments, while the second complete method *(C2)* computes increments with a single query (Figure 10).

1. *What **is a thing**?*
2. *What **is a foaf:Person**?*
3. *What is a foaf:Person and has voc:country **something**?*
4. *What is a foaf:Person and **has voc:country countries:US**?*
5. *What is a foaf:Person and has voc:country countries:US and is rev:reviewer of **something**?*
6. *What is a foaf:Person and has voc:country countries:US and is rev:reviewer of something that has rev:reviewFor **something**?*
7. *What is a foaf:Person and has voc:country countries:US and is rev:reviewer of something that has rev:reviewFor **a instance:ProductType238**?*
8. *What is a foaf:Person and has voc:country countries:US and is rev:reviewer of something that **has rev:reviewFor a instance:ProductType238**?*
9. *What is a foaf:Person and has voc:country countries:US and is rev:reviewer of **something that has rev:reviewFor a instance:ProductType238**?*
10. *What is a foaf:Person and has voc:country countries:US and **is rev:reviewer of something that has rev:reviewFor a instance:ProductType238**?*
11. *What is a foaf:Person and has voc:country countries:US and is rev:reviewer of something that has rev:reviewFor a instance:ProductType238 and has purldc:publisher **something**?*
12. *What is a foaf:Person and has voc:country countries:US and is rev:reviewer of something that has rev:reviewFor a instance:ProductType238 and **has purldc:publisher instance-Site1:RatingSite1**?*
13. *What is a foaf:Person and has voc:country countries:US and is rev:reviewer of something that has rev:reviewFor a instance:ProductType238 and has purldc:publisher instance-Site1:RatingSite1 and has foaf:name **something**?*

Figure 16: Navigation scenario from the initial step to: *What is a foaf:Person and has voc:country countries:US and is rev:reviewer of something that has rev:reviewFor a instance:ProductType238 and has purldc:publisher instance-Site1:RatingSite1 and has foaf:name **something**?*

6.1 Conditions

Experiments were conducted through a prototype implemented in Scala 2.9.2 on a 64-bit Unix machine with a 3.00GHz Intel Core 2 Duo processor and 3.8GB memory. The SPARQL endpoint used in the experiments, Fuseki[8], was runned on the same machine and queried by our prototype with Jena 2.10.0.

Since the CPU-time of the SPARQL endpoint was not accessible, we measured user times for the computation of values, classes, properties and inverse properties on the SPARQL endpoint executed on our machine. We also measured CPU-time of our prototype querying the SPARQL endpoint.

In order to have the least biased results, we restarted the computer for each experiment on a dataset with a different size. Moreover, on each dataset, first, Scalewelis restarded the SPARQL server for each method but did not restart it between steps of the scenario because the cache of the SPARQL endpoint within the scenario is used in real cases, it would be a bias to remove it in our experimentations. Second, the six methods were executed one after the other and, in order to smooth the experiments, the overall execution were evaluated ten times and we took the average.

6.2 Results

For each dataset, each navigation method and each navigation step, we measured increment times for values, classes, properties and inverse properties as well as increment count for classes, properties and inverse properties.

Along the overall navigation scenario, partial methods failed to compute all the increments only on step 6 of the scenario. On every other step and on every dataset, partial methods succeeds to compute all the increments, even the first partial method using only ten results to compute increments.

We formulate four hypotheses on the experiments. First hypothesis *(H1)*, if a partial method has a better accuracy than another, then this first partial method takes more time to compute increments than the other. Second hypothesis *(H2)*, the complete method *(C2)* using a single query to compute increments takes less time to compute increments than the complete method *(C1)* using one query per type of increment. Third hypothesis *(H3)*, partial navigation is sufficient to find increments. Fourth hypothesis *(H4)*, every partial navigation takes less time to compute increments than the faster complete method.

Section 6.2.1 discusses global times along the overall navigation scenario for each dataset and for each method. Section 6.2.2 discusses the initial step of the navigation in detail. Section 6.2.3 discusses usual steps of the navigation in detail. Finally, Section 6.2.4 discusses the validity of the four hypotheses.

[8] http://jena.apache.org/documentation/serving_data/

	Method	Values	Classes	Properties	InvProp	Increments	LIS Engine	Global
DS1	P1	1,095	0,967	0,150	0,251	2,463	0,114	**2,537**
	P2	1,311	1,203	0,328	0,399	3,241	0,245	**3,441**
	P3	1,359	1,806	0,734	0,736	4,635	0,802	**5,354**
	P4	1,684	2,686	1,808	1,659	7,837	2,735	**10,357**
	C1		2,174	2,757	8,576	13,507	0,081	**13,558**
	C2					20,649	0,044	**20,683**
DS2	P1	1,595	1,732	0,124	0,345	3,796	0,121	**3,879**
	P2	2,958	1,945	0,299	0,592	5,794	0,234	**5,977**
	P3	3,012	2,606	0,687	0,967	7,272	0,829	**8,006**
	P4	3,425	5,015	3,409	3,473	15,322	5,250	**20,022**
	C1		6,284	11,291	41,471	59,046	0,097	**59,116**
	C2					97,171	0,057	**97,222**
DS3	P1	1,474	2,995	0,133	0,353	4,955	0,121	**5,044**
	P2	4,381	3,232	0,287	1,154	9,054	0,260	**9,256**
	P3	7,843	3,908	0,805	1,528	14,084	1,042	**15,049**
	P4	8,362	7,100	3,962	4,852	24,276	6,926	**30,511**
	C1		14,109	31,210	269,521	314,840	0,101	**314,919**
	C2					550,758	0,068	**550,833**

Table 4: Total duration in second of the overall navigation scenario on the three datasets each with the six navigation methods

6.2.1 Overall Navigation Scenario

Results Computation times of the overall navigation scenario for each navigation method and for each dataset are summarized in Table 4. Increment times are the sum of value, class, property and inverse property times. There are blanks on value times for the two complete methods because those methods do not need to compute values in order to compute increments. There are also blanks on class, property and inverse property times for the complete method *(C2)* because this method computes all the increments in one single query. There is thus no information about the partition of the time of increments between classes, properties and inverse properties. Finally, global times are approximately the sum of both increments and LIS engine. In fact, increment times are user-times that the SPARQL endpoint takes to compute increments, LIS engine times are CPU-times of the whole process on our machine and global times are user times of the whole process.

Partition that are the collected increments for the overall navigation scenario excluding the initialization step is presented for the three datasets (Figure 17). The initialization step is excluded because this step computes only class increments even if there are properties and inverse properties, it would imbalance the result to incorporate the initialization step.

Interpretation Table 4 gives unexpected results for the comparison of the complete method *(C1)* with several SPARQL queries to the complete method with a single SPARQL query *(C2)*. We thought that using several SPARQL

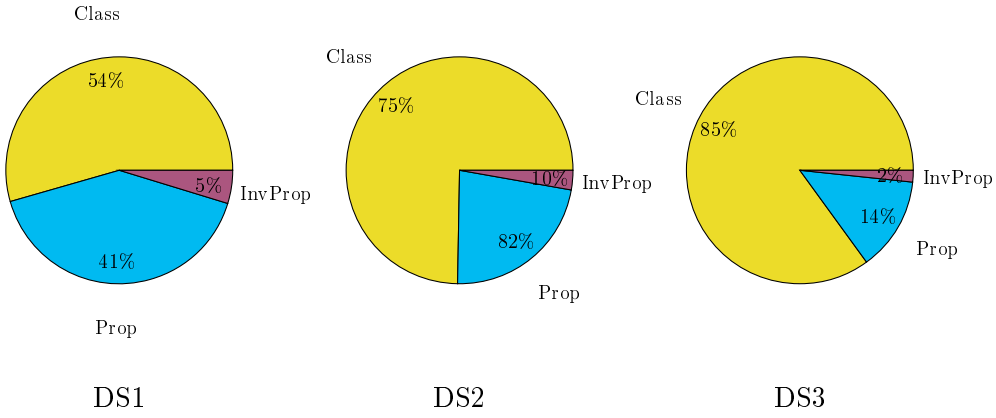

Figure 17: Partition of increments collected on the three datasets, for the overall navigation scenario excluding the first step

queries would be much slower than using a unique SPARQL query because in SPARQL queries of the complete method *(C1)*, the user query is expressed three times, but results say the complete method *(C1)* is approximately two times faster than the complete method *(C2)*. In the following, we only consider the complete method *(C1)* to compare to partial methods.

Global times of the complete method are approximately multiplied by a factor five from *(DS1)* to *(DS2)* and also from *(DS2)* to *(DS3)*. That is not the case for global times of partial methods that are not even multiplied by a factor two from *(DS1)* to *(DS2)* and also from *(DS2)* to *(DS3)*.

Global times of complete methods grow so fast because the computation time of inverse properties takes approximately 60 percent of the global time in *(DS1)*, 70 percent of the global time in *(DS2)* and 80 percent of the time in *(DS3)*. However, in partial methods, it is the computation of class increments that takes the most time. The latter behavior seems reasonable because the majority of collected increments during the navigation scenario are class increments (Figure 17). Furthermore, in Berlin benchmark, the bigger the dataset is, the more important the proportion of classes is, because only the number of classes grows when the dataset grows (Table 1).

The LIS engine we implemented takes with the partial method *(P4)* 5 seconds on *(DS2)* and 7 seconds on *(DS3)*. It represents one quarter of the global time and this time can not be ignored but there is little benefit to optimize it. We explain the increase of time of the LIS engine by partial queries it has to build from the results. In fact, no special efforts have been made to reduce this time.

	Classes
DS1	0.8
DS2	1.6
DS3	3.0

Table 5: Duration in seconds of class computation on the three datasets, for the first step of the navigation scenario: *What **is a thing**?*

6.2.2 Initialization Step

Results The first navigation step, *What **is a thing**?*, searches only for class increments and requires less than 1 second on *(DS1)*, less than 2 seconds on *(DS2)* and 3 seconds on *(DS3)* (Table 5).

Interpretation The computation of increments takes no more than a few seconds. Since this step initializes the search, the duration time is acceptable and does not need to be optimized.

6.2.3 Usual Steps

Results In usual steps, not like in the initialization step, the standard behavior of the methods takes place. We present in detail results from step 2, *What **is a foaf:Person**?*, because this step is the first usual step in the navigation scenario (Figure 18). We compare the evolution of the computation time first for each method on the three datasets. We also present in detail results from step 6, *What is a foaf:Person and has voc:country countries:US and is rev:reviewer of something that has rev:reviewFor **something**?*, because this step is the only step of the navigation in which partial methods did not find every possible increment (Figure 19). The projection is no the same as in the Figure of step 2, we compare the evolution of increments counts and times on the three datasets for each method. In step 6, property and inverse property increments are computed by every partial method. Only classes increments are not computed by partial methods. In fact, classes in step 6 are used as values, they are product types. This explains the important number of classes, their augmentation with a bigger dataset and the difficulties to find every class increments with partial methods.

For each dataset and each navigation method, the computation time of increments is presented in the axis growing upward and the number of valid increments is presented in the axis growing downward. Each bar of the diagram is partitioned into several components: those components are either values, classes, properties, inverse properties, increments or LIS engine.

Interpretation There is a total of 8 increments in step 2: 1 class, 7 properties and 1 inverse property. All the increments are found by every method.

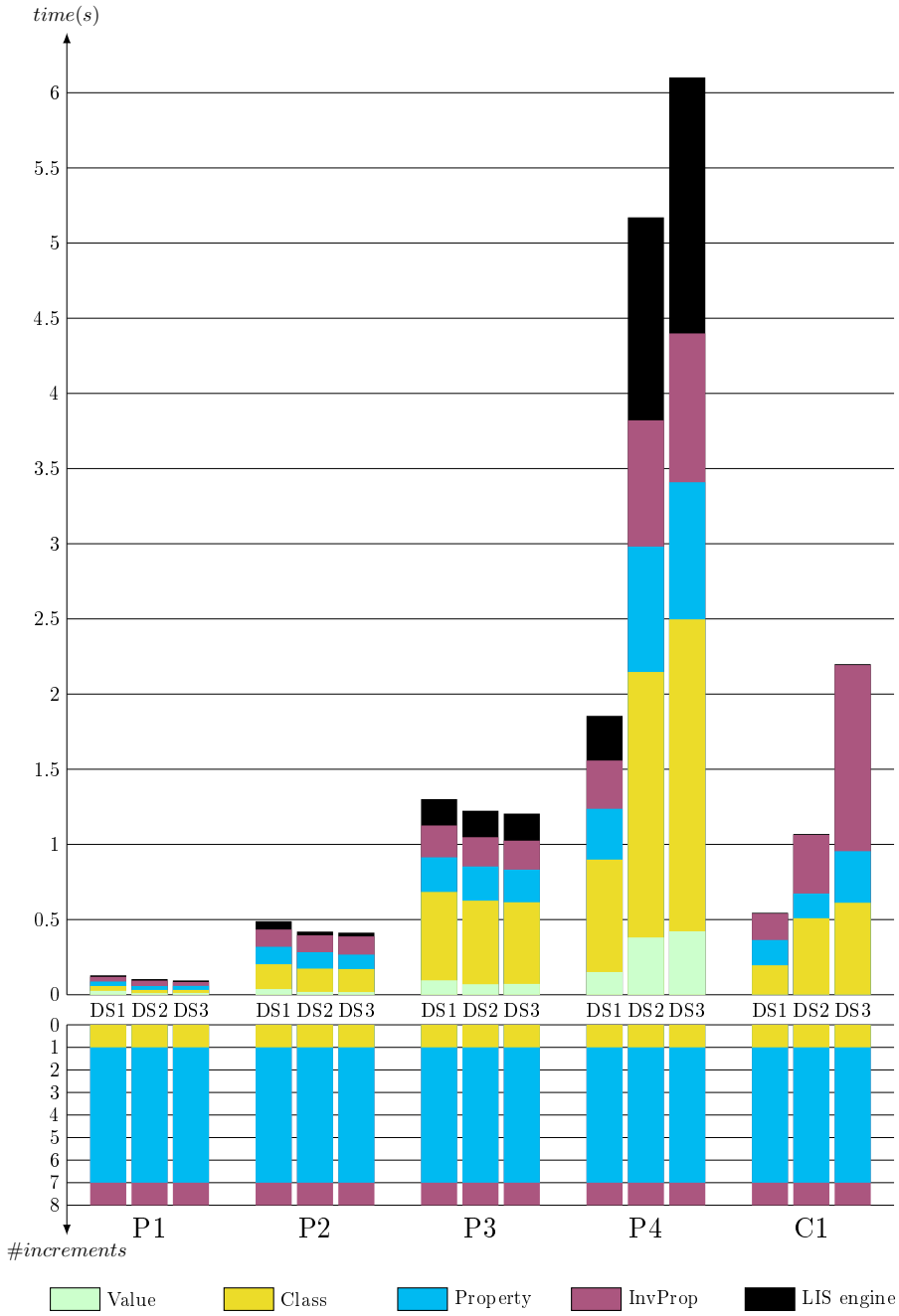

Figure 18: Increment counts and times with the six navigation methods each on the three datasets, for step 2 of the navigation scenario: *What* ***is a foaf:Person****?*

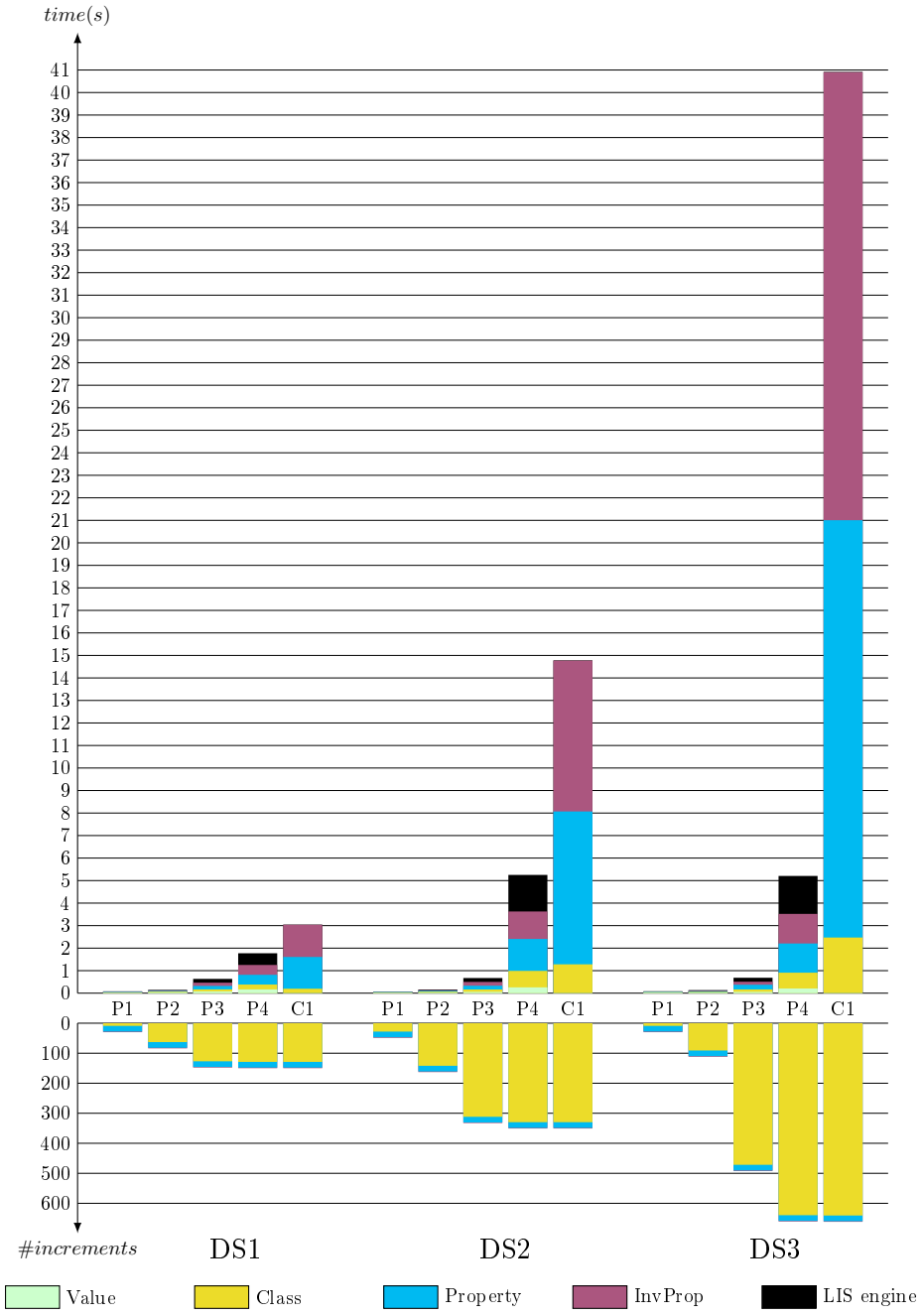

Figure 19: Increment counts and times on the three datasets each with the six navigation methods, for step 6 of the navigation scenario: *What is a foaf:Person and has voc:country countries:US and is rev:reviewer of something that has rev:reviewFor* ***something****?*

In step 6, the bigger the dataset is, the more valid increments there are because there are more available classes in bigger datasets. In fact, there are 19 valid properties and 2 valid inverse properties on every dataset, but there are 129 valid classes on *(DS1)*, 330 valid classes on *(DS2)* and 641 valid classes on *(DS3)*. Results of the query are products. Some partial methods do not collect all the valid increments. In fact, the bigger the dataset is, the less partial methods there are that compute all the increments. The partial method *(P4)* is the only partial method that collects every increment in every dataset.

The difference between the four partial methods is that the more values the partial method uses, the more time the partial method takes to compute increments. The first three partial methods *(P1, P2 and P3)* each keep a similar computation time whatever the dataset. Their computation time is even smaller if the dataset is bigger, it is probably because of the case used in the endpoint. It is not the case for the most accurate partial method *(P4)* that needs more time if the dataset contains more triples. However, the required computation time for partial methods in step 2 is similar to the required computation time for partial methods in step 6. That is not the case with the complete method, its computation time in step 6 is far bigger than in step 2. Finally, the partition of the increments is approximately respected in every partial method, except for the LIS engine that requires proportionally more time in the partial method *(P4)*.

The complete method takes two or three times less time than the partial method *(P4)* in step 2. However, on step 6, the complete method takes far mote time than *(P4)*, especially on the two bigger datasets. Moreover, that is surprising that the complete method uses so much time to find property and inverse property increments since there are so few of them to find, even the partial method *(P1)* find them with an accuracy of ten values.

6.2.4 Hypotheses Validation

Experiments on datasets created from Berlin SPARQL Benchmark allows us to validate or reject the four hypotheses. *(H1)* is validated, the computation time of partial methods grows more slowly than the size of data. *(DS3)* is 3 times the size of *(DS1)* whereas the computation time of partial methods on *(DS1)* is more than 2 times and less than 3 times the computation time of partial methods on *(DS3)*. *(H2)* is rejected, the complete method *(C1)* with one query per type of increment is much faster to the complete method *(C2)* with a single query for all the increments. *(H3)* is validated, partial methods are sufficient to compute increments. *(H4)* is validated, the partial methods taking the most time *(P4)* takes less time to compute increments than the complete method *(C1)*.

Finally, if we have to pick one method over all the proposed ones, we choose the partial method *(P3)* that is a little more slower than the partial

method *(P1)* but that collects almost every valid increment. Moreover, we can mix the two partial methods *(P1)* and *(P3)* with the following computation program composed of two steps. First, the partial method *(P1)* computes increments so that the user wait few time. Second, while the user is checking for increments, the partial method *(P3)* computes increments in order to provides the majority of accessible increments after some more time. From the results, even the partial method *(P1)* computes a lot of increments so that the user have something to wait until increments from the partial method *(P3)* are available.

6.3 Usability Study

We tested Scalewelis on a challenge about question answering over linked data: the QALD-3 challenge[9]. Moreover, another system of the LIS team, squall2sparql [8], was challenged on the same task[10]. squall2sparql implements a surjective mapping from a completely natural but restricted language, SQUALL, to SPARQL. The task was as follows:

> Given a RDF dataset and a natural language question or set of keywords in one of six languages (English, Spanish, German, Italian, French, Dutch), either return the correct answers, or a SPARQL query that retrieves these answers.

We challenged Scalewelis to answer ninety-nine questions. Organizers of the challenge prepared a special SPARQL endpoint for the occasion. This endpoint contained only the English version of DBpedia 3.8. Unfortunately, Scalewelis could not use this endpoint because response times were not practicable. We could not even get all the classes to begin the navigation. So, we connected Scalewelis on the standard DBpedia SPARQL endpoint[11]. In order to explore DBpedia, we chose to setup Scalewelis to navigate partially with a thousand results to approximate the computation of increments. Consequently, answer time was acceptable in general.

Scalewelis was piloted by us to answer to the questions, both our system and our own capabilities to find answers explain our results. By contrast, SWIP [20] that was also challenged on the same task, requires a full natural group of words as input and transforms it to SPARQL so that human intervention is reduced. We had, for each question, to navigate from the initial step, *What **is a thing**?*, to the focus that gives the answer to the question. We had to find correct formulations and correct increments to add to the focus in order to get correct answers. For example, with the following question,

[9] Question Answering for Linked Data
http://greententacle.techfak.uni-bielefeld.de/ cunger/qald/index.php?x=task1&q=3

[10] This is the first time the LIS team participated to the challenge

[11] http://dbpedia.org/sparql/

	Total	Processed	Right	Partially	Recall	Precision	F-measure
squall2sparql	99	96	77[1]	13	0.85	0.89	**0.87**
CASIA	99	52	29[4]	8	0.36	0.35	**0.36**
Scalewelis	99	70	32[2]	1	0.33	0.33	**0.33**
RTV	99	55	30[3]	4	0.34	0.32	**0.33**
Intui2	99	99	28[5]	4	0.31	0.31	**0.31**
SWIP	99	21	14[6]	2	0.15	0.16	**0.16**

Table 6: QALD-3 evaluation results for DBpedia

How many people live in the capital of Australia?, we first searched for cities, then we selected the property *capital of* and the resource *Australia*. Finally, to answer the question, we selected the property *population* and we picked up the answer. Otherwise, there was another faster solution to answer to this question, by selecting directly the capital of Australia by its name, and by choosing the property *population*. This second method was sometimes needed because the first method suffered, in some case, from computational difficulties. We shot down the navigation when computation of increments needed more than one minute.

When we began the challenge, we were not able to answer with the second method presented above. LISQL2 did not allow to use a URI as subject of the sentence. LISQL2 provided instead a special query that asked a description for this URI. This description was a table of properties and values corresponding to the URI, but to find information in this table needed too much time and effort.

We were only able to answer 70 questions over the 99 questions, requiring few minutes per question. There were two kinds of questions we did not answer: first, questions involving quantities like: *Which German cities have more than 250000 inhabitants?* because we did not implement filters on Scalewelis and, second, questions for which we were not able to find the increments to answer, like: *Give me all B-sides of the Ramones.*, because of human limitations or because the increment was not listed due to the partial navigation.

The score of Scalewelis is compared to the other challengers (Table 6). Scalewelis ranked third out of six candidates with 38 correct answers plus 1 partial answer. If systems were ranked on correct answers, Scalewelis would have ranked second. Concerning the two systems we discussed above, squall2sparql ranked first, whereas SWIP ranked last. Those results will be presented in a workshop, we are currently writing a paper to submit about Scalewelis.

Scalewelis was theoretically able to correctly answer to the 70 questions we chose to answer, but we failed to answer 37 of them choosing incorrect increments to add to the query because of three main reasons. First, English is not our mother tongue, incorrect increments were used instead of correct

ones because of misunderstandings. Second, we had sometime the choice between two increments differing only on their prefixes, for example we had sometimes to choose a property with either a *dbpedia ontology* prefix or a *dbpedia property* prefix. Third, the standard DBpedia SPARQL endpoint to which Scalewelis was connected added noise to the search compared to the special SPARQL endpoint.

7 Conclusion and Perspectives

We presented a scalable query-based search system that provides a guided and expressive semantic search over SPARQL endpoints, Scalewelis. We took inspiration from Sewelis and we explored improvements, mostly about scalability issues. First, we adapted the LISQL language to become more natural to conduct the search with a formal but still natural language. Second, we defined a SPARQL semantics to LISQL2 so that Scalewelis can connect to SPARQL endpoints. Data independence is achieved and Scalewelis can be used on existing datasets that are maintained by other sources. Third, we formalized the notion of focus in LISQL2 so that it supports inductive definitions over the incremental construction of queries. Fourth, partial methods to compute increments were introduced and compared to complete ones. We experimented those methods on datasets with similar structures but different sizes thanks to the Berlin SPARQL Benchmark. It resulted that partial methods are sufficient to compute most increments. Their computation time is sub-linear in the size of data, and is smaller than for complete methods. Finally, in order to test our system on a real and large dataset, we took part in the QALD-3 challenge on DBpedia (2 billion of triples). We were able to answer to 70 questions out of 99 with acceptable response times.

There are some perspectives we thought about but we did not explore. First, from SPARQL, there are missing functionalities such as comparisons or aggregations. It is another challenge to adapt the full expressivity of SPARQL into a scalable system providing usable navigation.

Second, We did not test our hypotheses (Section 6.2) on real datasets, for example on DBpedia. Tests on real datasets would show the importance of the schema in order to compute increments. We also did not conduct user studies to evaluate the real usability or our Scalewelis.

Third, In order to compute increments with a partial method, we limited results on focus to the first values returned by the SPARQL endpoint (Figure 11). However, we do not know how representatives those values are compared to all the results. Instead of taking the first values to compute increments, we could take random values to better approximate all the results. Moreover, when the engine uses a lot of values to compute increments, the computation time of the LIS engine grows considerably and takes significant time. Investigations on the LIS engine could explain where the code could

be optimized in order to reduce this computation time. Concerning optimization to compute increments, statistics on data could be made in order to initialize the system with pre-computations and speed-up computation at utilization.

Finally, some efforts can be made to define a mechanism of automatic completion for keyword search. The completion would be used when searching for results because all the results are not presented to users. The completion would also be used when searching for increments that are not visible with partial computation.

References

[1] SPARQL query language for RDF. preprint, available at http://www.w3.org/TR/sparql11-query/.

[2] R. Angles and C. Gutierrez. The expressive power of SPARQL. In A. P. S. et al, editor, *International Semantic Web Conference*, Lecture Notes in Computer Science, pages 114–129. Springer, 2008.

[3] H. P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[4] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. DBpedia - a crystallization point for the Web of Data. *J. Web Sem.*, 7(3):154–165, 2009.

[5] C. Bizer and A. Schultz. The Berlin SPARQL Benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):1–24, 2009.

[6] D. Damljanovic, M. Agatonovic, and H. Cunningham. FREyA: An interactive way of querying linked data using natural language. In *ESWC Workshops*, volume 7117 of *Lecture Notes in Computer Science*, pages 125–138. Springer, 2011.

[7] L. Ding, T. W. Finin, A. Joshi, R. Pan, R. S. Cost, Y. Peng, P. Reddivari, V. Doshi, and J. Sachs. Swoogle: a search and metadata engine for the semantic web. In D. A. G. et al, editor, *CIKM*, pages 652–659. ACM, 2004.

[8] S. Ferré. Squall: A controlled natural language for querying and updating rdf graphs. In T. Kuhn and N. E. Fuchs, editors, *CNL*, Lecture Notes in Computer Science, pages 11–25. Springer, 2012.

[9] S. Ferré and A. Hermann. Reconciling faceted search and query languages for the semantic web. *IJMSO*, 7(1):37–54, 2012.

[10] A. Harth. VisiNav: A system for visual search and navigation on web data. *J. Web Sem.*, 8(4):348–354, 2010.

[11] Philipp Heim, Thomas Ertl, and Jürgen Ziegler. Facet Graphs: Complex semantic querying made easy. In L. A. et al, editor, *ESWC (1)*, Lecture Notes in Computer Science, pages 288–302. Springer, 2010.

[12] P. Hitzler, M. Krötzsch, and S. Rudolph. *Foundations of Semantic Web Technologies*. Chapman & Hall/CRC, 2009.

[13] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. YAGO2: A spatially and temporally enhanced knowledge base from wikipedia. *Artif. Intell.*, 194:28–61, 2013.

[14] E. Kaufmann and A. Bernstein. Evaluating the usability of natural language query languages and interfaces to semantic web knowledge bases. *J. Web Sem.*, 8(4):377–393, 2010.

[15] Y. Lei, V. S. Uren, and E. Motta. Semsearch: A search engine for the semantic web. In S. S. et al, editor, *EKAW*, Lecture Notes in Computer Science, pages 238–245. Springer, 2006.

[16] V. Lopez, M. Fernández, E. Motta, and N. Stieler. PowerAqua: Supporting users in querying and exploring the semantic web. *Semantic Web*, 3(3):249–265, 2012.

[17] E. Mäkelä, E. Hyvönen, and S. Saarela. Ontogator - a semantic view-based search engine service for web applications. In I. F. C. et al, editor, *International Semantic Web Conference*, Lecture Notes in Computer Science, pages 847–860. Springer, 2006.

[18] B. Moseley and P. Marks. Out of the tar pit. preprint (2006), available at http://shaffner.us/cs/papers/tarpit.pdf.

[19] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical report, Citeseer, 2004.

[20] C. Pradel. Allowing end users to query graph-based knowledge bases. In A. t. T. et al, editor, *EKAW*, Lecture Notes in Computer Science, pages 8–15. Springer, 2012.

[21] G. M. Sacco and Y. Tzitzikas, editors. *Dynamic taxonomies and faceted search*. The information retrieval series. Springer, 2009.

[22] G. Smith, M. Czerwinski, B. Meyers, D. C. Robbins, G. G. Robertson, and D. S. Tan. FacetMap: A scalable search and browse visualization. *IEEE Trans. Vis. Comput. Graph.*, 12(5):797–804, 2006.

[23] Osma Suominen, Kim Viljanen, and Eero Hyvönen. User-centric faceted search for semantic portals. In E. F. et al, editor, *ESWC*, Lecture Notes in Computer Science, pages 356–370. Springer, 2007.

[24] H. Wang, Q. Liu, T. Penin, L. Fu, L. Zhang, T. Tran, Y. Yu, and Y. Pan. Semplore: A scalable IR approach to search the Web of Data. *J. Web Sem.*, 7(3):177–188, 2009.